A Framework for Variable Content Document Generation with Multiple Actors

Abel Gómez, M. Carmen Penadés, José H. Canós, Marcos R. S. Borges, Manuel Llavador

A Framework for Variable Content Document Generation with Multiple Actors

Abel Gómez[a,*], M. Carmen Penadés[b], José H. Canós[b], Marcos R. S. Borges[c], Manuel Llavador[d]

[a] *AtlanMod, École des Mines de Nantes - INRIA - LINA*
4 rue Alfred Kastler. 44307 Nantes, France
[b] *ISSI-DSIC. Universitat Politècnica de València.*
Cno. de Vera, s/n. 46022 Valencia. Spain
[c] *Programa de Pós Graduação em Informática. Departamento de Ciência da Computação. Instituto de Matemática. Universidade Federal do Rio De Janeiro. Brazil*
[d] *Instituto Tecnológico de Informática. Universitat Politècnica de València.*
Cno. de Vera, s/n. 46022 Valencia. Spain

Abstract

Context: Advances in customization have highlighted the need for tools supporting variable content document management and generation in many domains. Current tools allow the generation of highly customized documents that are variable in both content and layout. However, most frameworks are technology-oriented, and their use requires advanced skills in implementation-related tools, which means their use by end users (i.e. document designers) is severely limited.

Objective: Starting from past and current trends for customized document authoring, our goal is to provide a document generation alternative in which variants are specified at a high level of abstraction and content reuse can be maximized in high variability scenarios.

Method: Based on our experience in Document Engineering, we identified areas in the variable content document management and generation

*Corresponding author. Address: AtlanMod, École des Mines de Nantes. 4 rue Alfred Kastler. 44307 Nantes, France. Tel: +33 2 51 85 82 39

Email addresses: abel.gomez-llana@inria.fr (Abel Gómez), mpenades@dsic.upv.es (M. Carmen Penadés), jhcanos@dsic.upv.es (José H. Canós), mborges@nce.ufrj.br (Marcos R. S. Borges), mllavador@iti.upv.es (Manuel Llavador)

field open to further improvement. We first classified the primary sources of variability in document composition processes and then developed a methodology, which we called DPL – based on Software Product Lines principles – to support document generation in high variability scenarios.

Results: In order to validate the applicability of our methodology we implemented a tool – DPLFW – to carry out DPL processes. After using this in different scenarios, we compared our proposal with other state-of-the-art tools for variable content document management and generation.

Conclusion: The DPLFW showed a good capacity for the automatic generation of variable content documents equal to or in some cases surpassing other currently available approaches. To the best of our knowledge, DPLFW is the only framework that combines variable content and document workflow facilities, easing the generation of variable content documents in which multiple actors play different roles.

Keywords: Variable Data Printing, Document Product Line, Feature Modeling, Model Driven Engineering, DITA, Document Workflow, Organizational Modeling, Document Generation

---

1. Introduction

Managing variable content documents is a key aspect in domains such as e-learning, e-commerce, e-government and software development. The main challenges in generating customized manuals, contracts and governmental documents, among others, are the processes that involve defining document variants and content reuse [1, 2]. Although the documents usually follow a standard structure, they include some sections that can be repeated across different documents and others that are specific to a particular case. Although supporting variable content makes document generation more efficient it requires methods of specifying and handling variations.

In the Document Engineering field the problem of generating customized documents is known as Variable Data Printing (VDP). Increasingly sophisticated approaches to VDP have been proposed in recent decades, ranging from personalized letters in the Mail Merge [3] style, to customized multimedia documents [1, 2, 4, 5, 6]. Traditional approaches dealt with variability by including form fields in the documents, so that user-provided values become part of the document. More recent approaches have gone beyond variable data to support variability in document contents, but most of them require

document designers to be experts in XML and associated technologies such as XSLT (e.g. the *Document Description Framework*, DDF [1]), which they frequently lack. The challenge is then to provide a powerful document generation alternative that hides the complexity intrinsic to high variability scenarios. It is therefore a natural solution to develop end-user tools that move the definition of variability closer to the problem domain, to give users the option of distinguishing between the fixed and variable parts of the document, regardless of the final document format.

However, disguising the technical complexity is not the only problem to be solved. Document generation processes are complex activities that involve different participants, each with different tasks and different access rights to certain parts of the document. A description of the actors, responsibilities and access rights within a document generation process (generally known as the specification of the *Document Workflow*) is therefore required. Support for document workflow definition and enactment is mandatory for any tool aiming at providing organization-level document generation and management. Instead of isolated editing actions requiring manual synchronization to produce a final document, document workflow-enabled environments provide uniformity and global management of document creation processes.

To sum up, end-user orientation must be a distinguishing feature of VDP tools. This means that (i) the entities handled with the tools must be close to the problem domain, and (ii) some methodological guidance supporting the specification of both the document content and the document workflow should be provided.

The Document Product Lines (DPL) approach [7] provides a framework for variable content document generation that follows an alternate path to the traditional variable document generation. DPL was created with a twofold goal: first, to make creating variable content documents available to non-experts by including a domain engineering process previous to the document generation itself; and secondly, to enforce content reuse at domain level. Both goals can be achieved under the principles of Software Product Line Engineering (SPLE). The key to the success of a DPL process lies in the definition of the variability model, which describes how documents can vary (the so-called *feature models)*, and in the existence of an organized collection of document components (*core assets*). The document components are pieces of content that can be combined to produce the final document using a customized document editor generated by the product line and which implements the document workflow.

In [8] we introduced DPLFW, a DPL implementation based on Model Driven Engineering (MDE) principles [9]. DPLFW supports all DPL processes and has been applied in different domains such as generating Emergency Plans [8] and technical documentation [10]. In this paper we extend the description given in [8] to (i) include validation mechanisms, (ii) add organizational and workflow modeling capabilities, and (iii) provide workflow enactment capabilities by using custom document editors built at runtime. All these features are illustrated in a complete and comprehensive example, and the incorporated improvements are shown by comparing our approach with other variable content frameworks.

This paper is organized as follows. Section 2 presents the different types of variable content scenarios. Section 3 introduces DPL, showing how SPLE techniques have been adapted to support variability-driven document generation. Section 4 introduces the DPLFW framework. Section 5 describes in detail the different components of the DPLFW framework, shows how they work by giving a practical example. Section 6 summarizes the main contributions of our proposal, and provides a comparison with other variable content document generation frameworks. Finally, Section 7 closes the paper by presenting our conclusions and outlining future work.

2. Motivation

Customization has been a recurrent issue in the Document Engineering field for decades. From the early SGML times until the highly customizable documents generated with the newest tools, a number of issues have been addressed by researchers and practitioners. In this section, we provide examples of different types of variable content scenarios. These examples will help us to put our work in context and clarify our present contribution to the field.

Example 1 (Presentation) *The publisher of a bestselling author has a publishing policy that includes the generation of different quality editions of a given book. Initially, a luxury edition is produced, followed weeks later by a hard cover version, which is in turn followed by a pocket edition at a much lower price. For each edition, a different design (i.e. page layout, font, illustrations, etc.) must be applied to the original text. If possible, automatic procedures should be used to generate the different versions.*

This is perhaps the very first case of variable content scenario and its solution has also been in place for a long time. The early markup languages such as SGML [11] prescribed the separation of content and layout in structured documents, using tags to delimit relevant content sections and style sheets to define sets of presentation instructions for different versions of the same content. The invention of the Web and HTML, a derivative of SGML, made markup languages very popular. Style sheets came to the Web later, when website designers realized that they had to fully rewrite the HTML code of their Web resources every time a site was restyled. Further developments led to the XML language and to XSL, the eXtensible Stylesheet Language as the framework supporting multiple views of structured documents. □

Example 2 (Variable data printing) *A large travel agency wants to advertise its new summer campaign. In order to increase customer satisfaction, they want to launch a new utility called "myAgent" that provides customized offers based on customer information. The offer is simply a document that includes personal customer data, plus a number of elements selected from a repository using as retrieval criteria a customer profile based on previous interactions with the system. With this type of service the agent hopes to avoid providing customers with information outside their scope of interest.*

This example (inspired by [1]) is a generalization of the classical Variable Data Printing (VDP) problem. Among the first systems supporting VDP, the Mail Merge utility allowed the design of document templates that included placeholders for parts of the document content. Batch processes were then launched in which different copies of the template were generated in such a way that placeholders were replaced by actual values from a database. Business companies were then able to generate customized documents such as the examples given above, as well as employee business cards and any type of document that took values from data sources. Different extensions of the basic model have been developed in recent decades. One of the latest solutions, the DDF [1], defined an XML based language for the specification of variable content based not only on database queries but on any expression that could even include some parts of the document located by means of XPATH expressions. As one may expect, recent VDP tools are based on

XML and hence also support the variability in presentation described in Example 1. □

Example 3 (Document families) *The Spanish Civil Defense Authority issued regulations to oblige different types of organizations to be prepared for emergencies. These required that every public service organization must draw up an emergency plan that includes all the relevant information from preparedness to response. The basic content and structure of the emergency plan is provided in the form of a table of contents, and this basic plan structure works for most organizations. However, in some cases additional content must be included, such as in the case of nuclear plants, where the basic plan must be accompanied by an evacuation plan for all locations less than 30 km away. In many cases parts of the plan content (like standard response procedures, or technical information on fire extinguishers) will be common to more than one organization. The aim of the Authority is to provide a tool able to automatically generate a customized emergency plan template from the characterization of a given organization. This template will reuse as much content as possible from previously developed plans, thus saving time and money.*

The idea of entities sharing common features while differing in others is not exclusive to document engineering. As a matter of fact, the notion of product family had appeared much earlier in other fields. Particularly relevant in the case of software development, the term *program family* was coined by Parnas [12] and later included in the notion of Software Product Line [13]. The essence of Software Product Line Engineering is to model a family of software artifacts using a language able to distinguish commonalities and differences in the members of the family. From the variability specification and using components from a repository it is possible to achieve significant reductions in development time and high reuse ratios in some domains.

The product line approach looks very promising for the development of families of documents in cases with high content variability. Other domains in which a product line approach to customized document generation is useful are e-government, e-learning and, in general, domains with extensive content variability and reuse. □

Example 4 (Technology Variability) *A food website has a large collection of recipes after years of collecting and organizing information. To gain an advantage over its competitors the web owners want to make recipe downloading more flexible at different levels. Firstly, by allowing users to get customized recipes for a given dish using their own ingredients (to avoid unavailable ingredients). Secondly, by providing different type content for a given cooking step according to the user's abilities. For instance, the instructions for preparing a bolognese sauce can be provided in a video for beginners and in texts of varying complexity for average and expert cooks.*

This case illustrates what we call technological variability and is a variant of the scenarios described in Examples 1 and 3. A specific part of a document can be "filled" with different types of content, which can be selected by the user when configuring a specific member of a document family. This type of customization is one of the features of the DPL approach, as will be seen in Section 6. □

Example 5 (Multi-actor editing processes) *A software development company wants to increase the efficiency of the development of the technical documentation associated with its products. One of their major concerns is the management of document variants when the parts of the products change. To cope with this, they plan to follow a product line approach, as described in Examples 3 and 4. However, there is an additional requirement: the product line must comply with the company's document workflows. A document workflow model is the specification of the process followed in the development of a document. The process is described in terms of the activities performed, the control and data flow between these activities, and the actors performing them. In general, technical documentation development is performed by a hierarchical organization in which (parts of) the document contents are drawn up by editors and later approved by someone else. This is an important step in guaranteeing the quality of the generated document.*

In a product line environment, the document workflow cannot be defined in advance, since it is dependent on the parts of the content selected for a given member of the document family. The classical product line approaches must therefore be extended to include document workflow management.

A document workflow definition mechanism is the key to obtaining end-user oriented tools. From the experience gathered during decades of Workflow Management Technology, graph-based languages are those best suited to making process-oriented specifications [14, 15]. In the case of document workflows, each node in the graph will typically represent an editing action that must be performed by some actor at some point in the process. By editing action we mean either providing or modifying content, as well as approving or rejecting a particular content. For instance, different parts of a software manual may be written and approved by different members of the company. Similarly, the evacuation procedures in an emergency plan can be designed by specialized safety engineers and approved by civil defense agencies before their release; or in a typical e-learning scenario, document workflow consists of a teacher compiling an exam taken by a student and corrected by a third person. □

Examples 1 to 5 illustrate the different ways of understanding customization in document generation. As the first conclusion, we believe that dealing with variable content documents is a complex task and is difficult to solve at a low abstraction level. Product line approaches provide methodological support to help document engineers follow systematic processes, starting with explicit variability modeling and automating generation and enforcing reuse. Our second conclusion is that handling large collections of reusable content requires tool support. In the following sections, we introduce DPL, our methodological approach to variable content document generation, and its supporting tool, DPLFW. The above examples also serve to define the framework we use to compare DPLFW with other variable content support tools.

3. Document Product Lines

DPL aims to apply SPLE principles, techniques and tools to the generation of variable content documents involving multiple participants. DPL provides methodological guidelines to model the commonality and variability in a document family as a set of features. Such document features are assigned to the different actors that play different roles. A document workflow model is generated from the features selected for a specific document. The workflow model describes the tasks that each user must carry out to obtain the final document. To achieve this, different custom document editors (targeted at a specific user) are generated (re)using components and following a

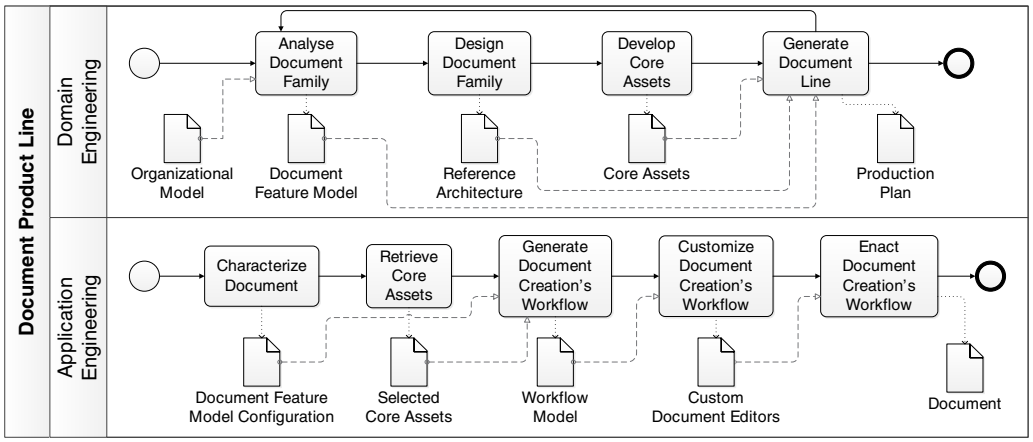


Figure 1: DPL-based Document Generation Process

product line approach. These editors implement the views of each actor in the workflow of creating the document, acting in a way similar to a wizard. The enactment of the workflow (i.e., execution by the editors) produces specific instances of the variable content document that will later be composed with a layout to generate a final version of the document. An outline of the DPL method is given in the remainder of this section.

The starting point of a DPL process is the identification of variability sources in documents from a domain-oriented perspective [16, 17]. To achieve this we adapted classical feature models [18] to the document generation domain. In this way we obtained an expressive method of defining variability and can take advantage of tools for analyzing such models [19, 20]. The adaptation was carried out bearing in mind the specific requirements of variable content documents.

Figure 1 summarizes the DPL document generation process. The process is described using the Business Process Modeling Notation (BPMN) [21]. Only the main tasks and artifacts are depicted for the sake of simplicity. Start and end events are represented by circles, data objects are shown as a sheet of paper, tasks by rounded boxes, sequence flows are shown as solid lines, message flows by broken lines and data associations are dotted lines. As in SPLE, the DPL process includes two iterative subprocesses. The first, called *Domain Engineering*, takes an organizational model as input and is composed of four tasks. In the *Analyse Document Family* task, a domain

engineer specifies the documents in terms of content and technology. The domain engineer must also identify the actors who contribute to the document and their specific responsibilities. These contributors are the members of the organization in which the DPL methodology is applied, and should be properly described in an organizational model. Since the organizational modeling stage is considered outside the scope of the DPL methodology, the organizational model is assumed to pre-exist. The result of the analysis is a document feature model including mandatory, optional and alternative features of both OR and XOR variants. Mandatory features are the parts that must be included in all the documents of the family, whereas the optional and alternative feature will only be included in certain members of the family. In the *Design Document Family* task, the generic document architecture is defined by identifying the document components (related to the content) and software components (related to content-supporting technology) required, according to the feature model built in the previous stage. Specific instances of the architecture are created later in the process, after the variability points for a specific document have been fixed.

DPL assumes the existence of a *Repository* where document and software components are stored and organized for reuse. They are the *core assets* in SPLE terminology. Metadata are attached to each core asset in order to support asset retrieval processes in the *Develop Core Asset* task to find existing components. If a requirement specified in the feature model cannot be fulfilled by any core asset in the repository, a new component should either be developed or retrieved from other repositories. In this case, a library of applications to create and/or modify core assets must be available. Finally, in the *Generate Document Line* task, a production plan is obtained. This is a process that specifies how the components are integrated according to the different relationships defined between the document features.

The second subprocess, called *Application Engineering*, supports the generation of the variable content documents by the collaboration of different actors. In the *Characterize Document* task, the document engineer (the person in charge of coordinating the creation of a specific document) selects the variability points, i.e. the optional and alternative features included in the document. This task includes the selection of both content and technology features. Next, the core assets are sought according to the variability specification made; and, in the *Generate Document Creation's Workflow* task, the assets are used to generate a *Workflow Model* which clearly defines how the document must be edited and completed by the different actors. This

workflow model can be customized by the document engineer, rearranging tasks and tuning actor responsibilities. The assets linked to the editing tasks include both the software components required to edit or to generate the final document and the document content components which pre-populate the editor's contents. These assets are used to generate the custom editors that represent a user-centered view of the document workflow. Finally, in the *Enact Document Creation's Workflow* task, the editors are used to complete, if necessary, the final content of the document. By final we mean that it will not contain any variable data since all the document components will have been instantiated and approved during the editing.

4. The DPLfw Framework

DPLfw provides the methodological and technological background to creating variable content documents by the DPL approach. DPLfw was developed following the MDE and Model Driven Architecture (MDA) [22] paradigms, which allowed us to take advantage of code generation techniques for the implementation of a tool prototype.

DPLfw was also designed to be extensible and highly configurable, allowing any new technology or platform to be plugged in. This requirement made us to choose Eclipse [23], one of the state-of-the-art development environments, for its creation. Three key technologies were selected for its implementation: the Equinox framework [24], the Eclipse Modeling Framework (EMF) [25, 26] and the Connected Data Objects (CDO) framework [27]. The first is an implementation of the OSGi standard [28], a dynamic component model and a service platform to build modular and extensible Java applications, which provides the basic runtime services for the Eclipse IDE itself. The second (EMF) is a framework to build applications using MDE techniques, raising the level of abstraction and reducing development time by using code generators. The third (CDO) is a framework built on top of EMF which allows concurrent and transactional modifications of distributed EMF models. CDO provides authentication, storage and retrieval mechanisms, regardless of the actual database management system used (the current version of DPLfw uses PostgreSQL [29] as its persistence back-end).

In some preliminary works [7, 30] we used the well-known *pure::variants* tool [31] to validate the DPL proposal. However, developing a full-featured framework to support DPL allows us to focus on the Document Engineering field and especially to incorporate our own feature metamodel tailored to its

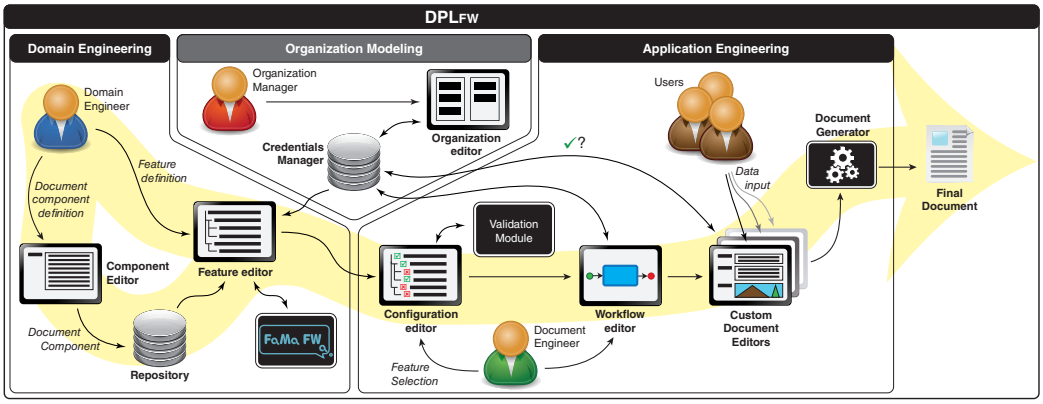


Figure 2: DPLFW overview

sources of variability, thus simplifying the complexities of generic SPL tools. This is demonstrated in the first fully functional Eclipse-based version of DPLFW, which was presented in [8]. However, this first version had two limitations; firstly, it only supported fully instantiated document components, whereas in practice things are somewhat more complicated. Some document components can only be partially instantiated, as in the case of templates, or form-based parts, which must be completed by the final users (the variable data). Secondly, no support for multiple actors was provided. However, the current version of DPLFW supports multiple actors, custom document editors and model validation.

The DPLFW is made up of a set of pluggable components – namely editors, viewers and explorers – which follow the Rich Client Platform [32] architecture. The implemented functionality facilitates exploring different repositories, defining organizations, creating new document components, defining document templates as feature models, tuning document workflow models, creating and displaying custom editors and generating documents. These elements communicate with a components repository and a credentials manager in a client-server architecture. In the remainder of this section we outline the main elements of DPLFW and how they communicate with each other.

Figure 2 describes how a fully-fledged DPL process is carried out in DPLFW. The *Domain Engineering* stage is an iterative process. For the sake of simplicity no specific order is enforced to execute its tasks as far as there is a fully populated document feature model describing the domain at the end of the stage. The *Feature Editor* is used by the *Domain Engineer* to

characterize the variability of the domain as a document feature model (as described in the *Analyse Document Family* task of the DPL process). The *Feature Editor* is closely related to three components. First, the *Credentials Manager* is a directory service which stores information about the members of an organization (users, groups, hierachy, login credentials, etc.). The organization's information can be edited by the *Organization Manager* using the *Organization Editor*. Secondly, the FaMa framework [33] is a validation and verification engine which uses formal representations to guarantee that the feature models defined in DPLfw have no errors. And thirdly, the *Repository* contains the core assets (document components) that will later be reused. All these elements support the *Analyse Document Family* task (cf. Figure 1). For the sake of simplicity, the *Reference Architecture* matches the structure of the feature model, and thus, the *Design Document Family* task is implicit and does not require user-interaction. The *Component Editor* supports the *Develop Core Assets* task and is used to create new document components and add them to the *Repository*. The *Generate Document Line*, which will describe how to retrieve and integrate the different components to obtain the final product, is also implicit: DPLfw implements automatic generation of the default production plan, since the structure of the document family is determined by the document feature model, every content component has a default handler (called *Disseminator*, see Section 5.3), and there exist predefined mechanisms to retrieve the different components from their corresponding repositories.

The remaining elements are related to the *Application Engineering* subprocess. The *Configuration Editor* supports the *Characterize Document* task through the selection of variability points. The *Document Engineer* is assisted in the configuration process by means of the *Validation Module*, avoiding configuration errors. Once a *document feature model configuration* is defined, DPLfw *Retrieves the Core Assets* of the selected features from the repository and *Generates a Document Creation Workflow* model automatically. This model contains explicit information about the tasks and the actors involved. These tasks are inferred from the domain specification made in the feature model, and can be fine-tuned using the *Workflow Editor* – as the *Customize Document Creation's Workflow* task specifies. Once a document workflow model is specified, the *Enact Document Creation's Workflow* task starts and the *Custom Document Editors* are generated by composing the document components. These editors present actor-specific views of the document based on the permission given to each participant in the document

workflow. These editors are in charge of controlling user privileges with regard to document content and are used to fill in any remaining variable data. Finally, the *Document Generator* integrates the different components to obtain a fully instantiated document generated in a specific format (printed, hypermedia, etc.).

5. A closer look at DPLfw

In this section we present a detailed description of all the components that were introduced in Section 4. For each component we give a detailed description of its design and implementation as well as a practical example of how it is used.

Our practical example is a software development organization which uses DPLfw and aims to generate customized software manuals targeted at different types of user. This organization is in charge of developing the DPLfw tool itself, and aims to use it to generate end-user manuals. The members of the organization (programmers, testers, managers, etc.) are grouped in different units: *Analysis & Design*, *Implementation*, *Testing*, *Deployment*, *Documentation* and *Project Management*. This organization has decided that the software manual of the DPLfw may be divided into four chapters, namely *Introduction*, *Installation*, *First Steps*, and *Version History*. However, they have detected that the contents of these chapters depend on two different factors: (i) the type of user and (ii) the DPLfw version for which the manual is generated. They have therefore decided to use the DPLfw to develop a family of software manuals.

5.1. The Organization and the Credentials Manager

Following classical workflow models, the different actors involved in the document generation process are specified according to an organizational model. Although the organization modeling stage is not considered as a part of a DPL process, as explained in Section 3, DPLfw requires a generic model which enables interoperability with existing organizational models.

Figure 3 shows the organizational model defined in DPLfw. It describes an organization as a hierarchy of actors. Actors may be individuals (users) or groups of users called units (e.g. departments). The users may belong to one or more units, and units may be composed of other units. Every unit is managed by one user. Actors are identified by a universally unique identifier (UUID), must have a name, and may have a description and an e-mail address

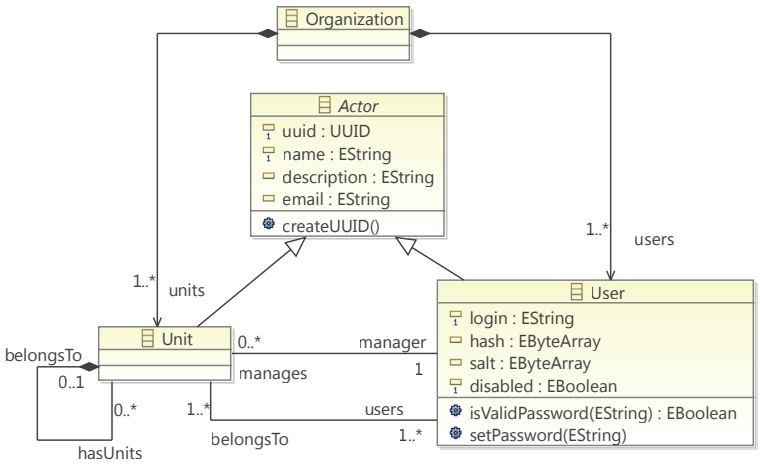


Figure 3: Organizational Model

(in the case of units, the e-mail address corresponds to a mailing list including the addresses of all the members). For users, the login information is also stored, i.e., a unique login alias, a (randomly salted) hashed password, and a disabled status flag.

DPLfw provides out-of-the-box support for simple organizational models which comply with this specification. They can be edited using the *Organization Editor* and may be stored either locally or remotely, following a client-server architecture. However, according to the idea that organizational modeling is not part of the DPL methodology, DPLfw can use any existing organizational model by means of the *Credentials Manager* service. Thanks to this service, the actual persistence format and location of the organizational models are transparent to the remaining DPLfw modules. The *Credentials Manager* provides an API to access different organization directory managers, whether they are built-in or the external (such as LDAP).

Example 6 *In our sample scenario, a software development organization produces customized manuals for its software products. The structure of this organization is shown in Figure 4, which shows what the organizational model editor looks like. In the screenshot users are grouped into the following units:* Analysis & Design, Implementation, Testing, Deployment – *which in turn includes a* Documentation *unit – and* Project Management. *Moreover, the* Deployment *unit is formed by* Analyst 1, Deployment Architect *(who is also the*

manager of the Unit), Documentation Manager*,* Programmer 1 *and* Tester 1.

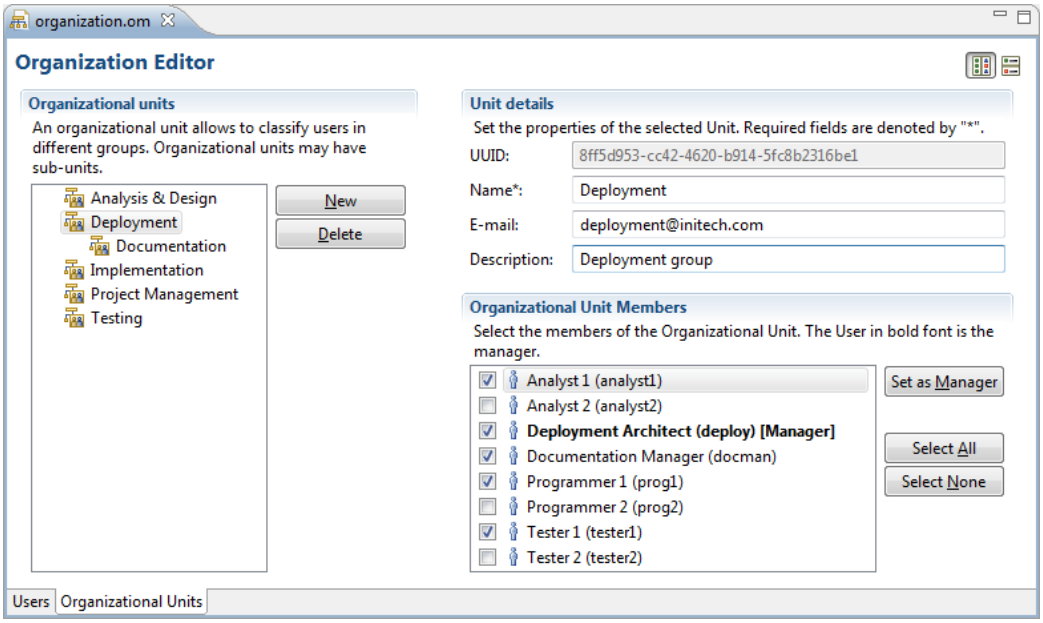


Figure 4: Organization Editor

5.2. The Repository

The automation of document generation processes relies heavily on the availability of the components that will be reused to build the different documents of the family. Such availability is granted by the Repository, which provides services for managing (i.e. creating, deleting and updating) and retrieving components (via e.g. keyword-based search). Additional services could be defined if required. We will now focus on document components (content components) for purposes of clarity.

The actual services provided by the Repository depend largely on the structure of the document components. In DPL, the document components of the Repository are called the *InfoElements* [7]. We modeled the Repository structure as shown in Figure 5: it is placed in a given location represented by a Uniform Resource Identifier (URI) [34] and may contain any number of *ResourceNodes*. These are elements which allow *InfoElements* to be organized

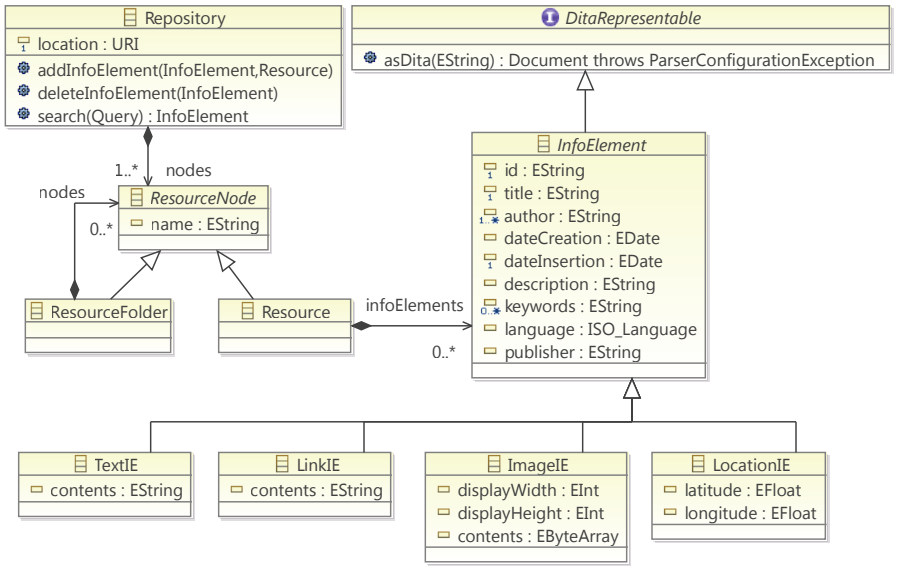


Figure 5: Repository Model

hierarchically. Two kinds of *ResourceNode*s can be defined: *ResourceFolders*, which may contain other *ResourceNode*s; and *Resources*, which contain the aforementioned *InfoElements* only. A *Repository* must also provide services for adding, removing and retrieving *InfoElements*, which are composed of two main blocks: data and metadata. The former is an encoding of the actual component content, be it text, image, or any other multimedia object. The latter is concerned with providing the information needed to describe and manage the content.

The set of attributes of the *InfoElement* class define the metadata schema used in DPL. Table 1 lists the metadata fields associated with *InfoElements*. We have selected a representative subset of the metadata elements defined in the Dublin Core Metadata Set [35]. The "Definition" column describes in a few words the meaning of the corresponding metadata "Element". "Multiplicity" indicates the maximum and minimum occurrences of each element in a metadata record. As can be seen in Table 1, *InfoElements* also have a type. For the sake of simplicity, in the current DPLfw version, types are implemented by inheritance. For instance, Figure 5 shows some different types of *InfoElements*, such as text (*TextIE*), link (*LinkIE*), image (*ImageIE*) and geographical coordinates (*LocationIE*).

Finally, *InfoElements* implement the *DitaRepresentable* interface to main-

Table 1: Repository metadata

Element	Definition	Multiplicity
Author(s)	Person/entity responsible of the *InfoElement*	1..*
Date of Creation	Date the *InfoElement* was created	0..1
Date of Insertion	Date the *InfoElement* was added to the repository	1
Description	An account of the *InfoElement*	0..1
Identifier	A unique identifier of the *InfoElement* in the repository	1
Subject	The main topic of the *InfoElement*	0..*
Keywords	Other topics of the *InfoElement*	0..*
Language	Language of the *InfoElement*	0..1
Publisher	Person/entity who distributes the *InfoElement*	0..1
Title	A name given to the *InfoElement*	1
Type	The nature or genre of the *InfoElement*	1

tain backwards compatibility with previous work [7, 30] and allows the use of automated tools for document generation, since *InfoElements* are represented and managed using the DITA standard [36]. DITA is an XML framework for the production of topic-oriented documentation. The main element of the DITA specification is the *topic*. DITA topics are organized into different hierarchies for different output documents, or DITA maps. In DPLfw an *InfoElement* is represented as a DITA topic.

Example 7 *Figures 6 and 7 show the user interfaces of the DPLfw components used to manage and develop the core assets. These screenshots show how to add new contents to a specific repository, in this case, components of the DPLfw manual. Figure 6 shows the* Repository Explorer*, which allows connecting to different repositories. Contents in repositories are organized hierarchically as described in Figure 5. When a specific repository is selected, users can add/edit/remove any content in it. New disseminators for different content types can easily be added to DPLfw following the OSGi architecture. Figure 7 shows how a new document fragment, an image* InfoElement *called* Project Preview*, is edited. Figure 7a shows the component metadata tab, and Figure 7b shows the content tab. The former is used to define the InfoElement's metadata and the latter is used to assign the content to the new document component. In this example, an image editor is used to select an image file. Other types of editors may be used to deal with other types of content (such as text).*

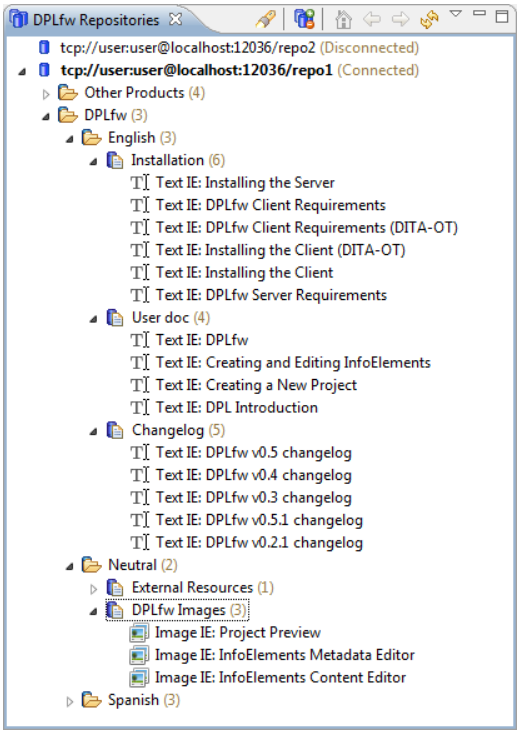
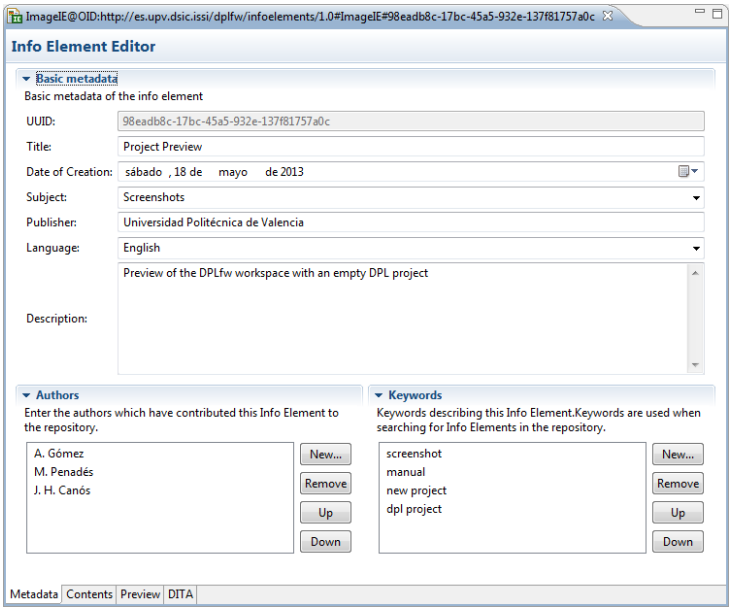
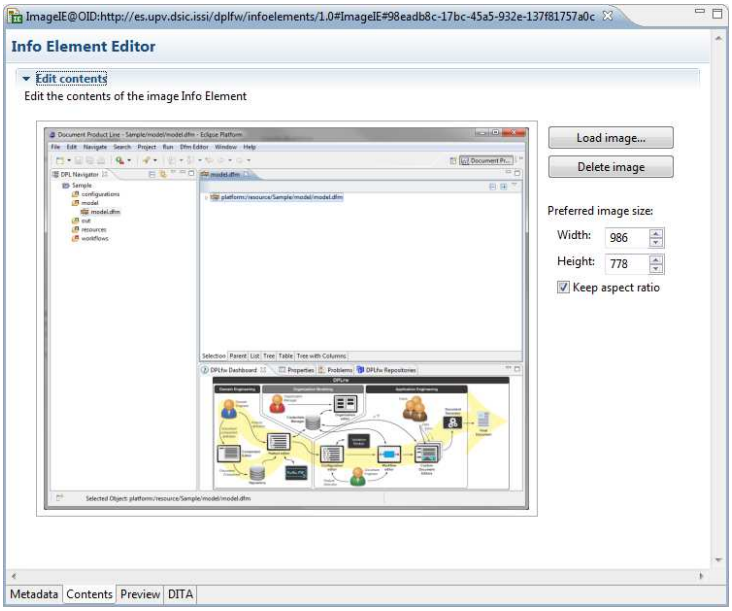


Figure 6: Repository Explorer



(a) Component Metadata Tab



(b) Component Contents Tab

Figure 7: Component Editor

5.3. The Feature Editor

In line with the rest of the Document Engineering community, we define a document as the union of two components: content and presentation. The document content includes a template that defines the logical structure of the document, plus the components that instantiate the template. The presentation includes the layout that defines exactly where each piece of content is to be placed and also how the piece will appear in the document. The latter is important because a given component may be shown in different ways. For instance, tax statements include some mandatory sections along with others that only apply to specific cases (*content variability*). Additionally, specific sections of a tax statement can be presented in different forms (parts of the tax statement can be produced as a printed document, keeping others only as a set of electronic forms). This technology dimension is relevant in the DPL process, since the specification of variability in a document family is done in terms of both content and technology.

To cope with both sources of variability, DPL can handle two different types of features: those related to document content (*ContentDocumentFeature* or CDF) and those related to the technology used to represent the content (*TechnologyDocumentFeature* or TDF). A CDF represents a part of the document and can be associated with one or more TDFs. As in classical feature models, cross-tree relationships may link document features in DPL variability models, such as the "requires" and the "excludes" constraints. These constraints may be checked in subsequent development stages to ensure consistency in selecting features.

The *Feature Editor* enables Document Feature Models to be defined. Figure 8 shows the feature metamodel supported by DPLFW and how it has been adapted to deal with multiple actors. We used *pure::variants* [31] as a reference in developing this metamodel, since it is widely known and used by the SPLE community. In DPLFW, a *DocumentFeatureModel* is composed of a set of *DocumentFeature* elements, which, as explained in Section 3, can be related to either content (*ContentDocumentFeature*) or technology (*TechnologyDocumentFeature*). A *DocumentFeature* can be declared as mandatory, optional, alternative (XOR group) or optionally selectable (OR group). Complex *Restrictions* (*requires*, *excludes* or logical combinations of these) may be defined between two or more features.

The ability to define different types of relationships and restrictions among features may bring great complexity into DPLFW feature models. In this context, due to the nature of feature models, the use of logic representations

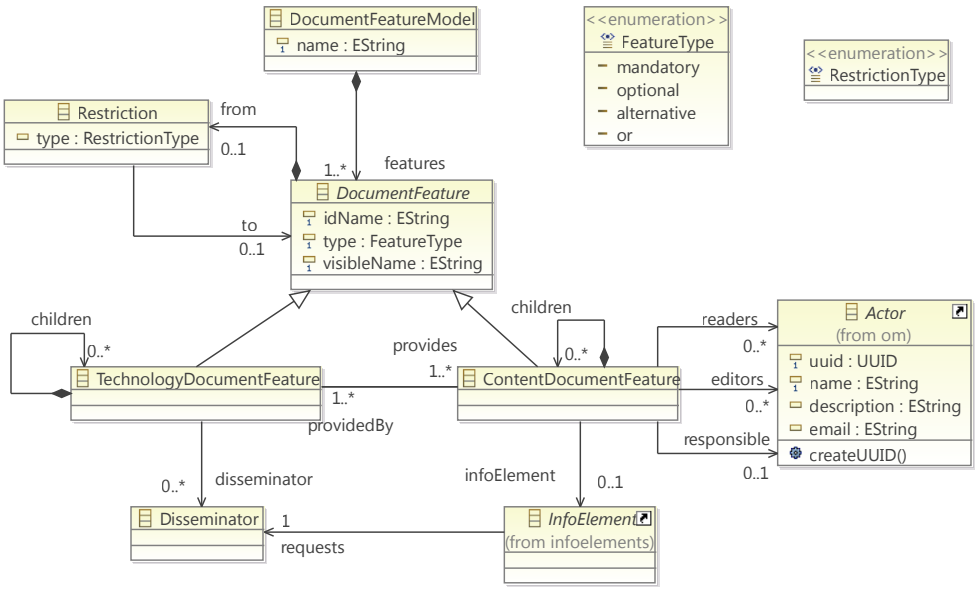


Figure 8: Feature metamodel

and languages can help in the development of model checkers. In line with this idea, the *Feature Editor* relies on FAMA [33], a framework which fulfills our requirements to detect and avoid errors in the definition of feature models. FAMA was integrated as a new module in DPLFW and its use is transparent to the user, hiding the complexities of the underlying formalism – Constraint Satisfaction Problems (CSP) [37] in the case of DPLFW. Using FAMA, DPLFW is able to detect different types of error (void feature models, dead features, false-mandatory features, etc. [38]) and provide detailed messages to help fix them easily. Actual document content is associated with content document features via instances of the *InfoElement* class.

In order to link actors with editing tasks, we have merged the organizational model with the DPL document feature metamodel (only the class involved, i.e. *Actor*, is represented for purposes of clarity). Both models are connected via three associations between the CDF and *Actor* classes. The instances of the *Actor* class will contribute to complete the *InfoElement* associated with the CDF with different roles: as an editor, an actor has read/write permissions; as a reader, he/she has only read permission; and, as the person responsible, he/she is responsible for approving the content. Only actors granted with one or more of these authorizations can access the

InfoElement associated with a CDF. The $0..n$ multiplicity in the *Actor* ends of the associations means that some CDFs can be non-editable, non-readable and/or do not require approval.

Finally, according to [39], a *Disseminator* represents a software component used to visualize different types of *InfoElements*. Examples of disseminators are text editors, image viewers, video players, and web services.

A document family is an instance of the above metamodel. The elements that compose this instance drive the definition of the architecture of the associated document editors that will be generated in the product line.

Example 8 *As specified above, the DPLFW software manual may be structured in four chapters:* Introduction*,* Installation*,* First Steps*, and* Version History*. The structure and contents of the final document depend on two factors: (i) the type of user at which the manual is targeted, and (ii) the DPLFW version for which the manual is generated.*

We defined the feature model shown in Figure 9 bearing in mind these two sources of variability. The model specifies the family of documents to be generated as a set of features. An exclamation mark denotes mandatory features, a question mark optional features, a double-headed arrow alternative features and a cross is the symbol for OR groups. The model contains four top-level CDFs – one for each of the chapters mentioned above – and one top-level TDF – which defines the type of manual.

*The first CDF (*Introduction*) is composed of one mandatory feature and two alternative features. The first represents the version of the DPLFW for which the manual will be generated and contains five alternative children features (from Version 0.2.1 to Version 0.5.1). The other two alternative features are the two different types of user at which the manual is targeted: system administrators (those in charge of installing and configuring the framework in a given organization) and end users (those who will use the tool to carry out DPL processes). The* Introduction *sub-features cover the two sources of variability we identified in the domain: version number and type of user. Since variations in the content of the document should depend on these factors, we introduced a set of restrictions ("requires" relationships). For example, a manual for system administrators must include general installation instructions, server instructions (the client does not need administrative rights to be installed), and the version history. A final users' manual only requires the first steps section. In both cases additional sections*

Figure 9: Feature Editor

may also be included, unless another restriction is violated.

The second top-level CDF, Installation*, has two mandatory sub-features:* Requirements *and* Installation Steps*. The former represents the software that must be installed and configured prior to installation of DPLfw components, while the latter describes the detailed steps to set-up an executable DPLfw environment. These two sub-features can be differentiated for both the server and the client part of the framework, i.e.,* Server Requirements*,* Client Requirements*,* Server Installation *and* Client Installation*. Prior to DPLfw version 0.5, the* DITA Open Toolkit *was required to be installed and configured as an external program; but in current versions of the framework a minimal runtime has been embedded in DPLfw itself. The features* DITA Open Toolkit *and* DITA Open Toolkit Install *are therefore optional. We also included some "requires" and "excludes" relationships to illustrate how DPLfw can manage different types of restrictions. The "requires" relationships lay down that if one of these sections is included, the other must be included too, and the "excludes" relationships state that no information about installing DITA must be included in the manual for the latest versions (i.e. v0.5.0 and v0.5.1). Additionally, the* Installation *feature has an optional sub-feature:* Installation Screencast*. Since this sub-feature represents a video, it requires a multimedia manual (represented by a TDF).*

First steps *is the third top-level CDF. It represents the sections of the manual which give instructions on how to begin to work with the tool. Tasks such as* Creating a New Project *or* Creating and Editing Infoelements *are covered in this part of the document. The optional children features (*Project Preview *and* InfoElements Editor Preview*) are contents that may enrich the document by showing screenshots.*

The last CDF is the Version History*, which contains a list of the bugs that have been solved in every release, together with new enhancements included in the different versions of the framework.*

As explained above, CDFs may be associated with actors. The actors' responsibilities/authorizations are set while the document feature model is built. As an example, we have defined the following authorizations in the sample scenario:

- *All the members of the organization have read permissions for the entire document.*
- *The* Introduction *CDF (and all sub-features) may be editable by the* Documentation *unit.*

- *The* Installation *CDF (and all sub-features) may be editable by the* Deployment *unit.*
- *The* First Steps *CDF will be unmodifiable. Its direct children will be editable by the* Documentation *unit, and the* Project Preview *and the* InfoElements Editor Preview *CDFs may be editable by both the* Documentation *and the* Implementation *units.*
- *The* Version History *will be editable by the* Implementation *unit.*
- *The* Version History *contents must be approved by* Programmer 1 *(the manager of the* Implementation *unit).*
- *The contents of all the remaining editable CDFs must be approved by the* Documentation Manager.

Finally, we modeled the TDF to cope with the diversity of formats in which the CDFs can be represented. In this case study, the DPLfw manual may be printed (for instance, a PDF file) or multimedia (an HTML web page with embedded video content). These options are modeled as alternative TDFs in Figure 9.

To illustrate the validation capabilities provided by FaMa, Figure 10 shows a modified version of the document feature model together with the Problems *view. There, we can see that an additional requirement has been added: the alternative feature* v0.5 *needs to have the* DITA Open Toolkit (ID: DitaOpenToolkit) *feature, as do Versions 0.2.1, 0.3 and 0.4. However, this restriction involves an error: the feature* v0.5 *(ID: V05) is a* dead feature*, i.e. it cannot be present in any possible product because it collides with the* DITA Open Toolkit *feature, which has an exclusion restriction. All these kinds of semantic errors, even if the feature model is void (i.e. there is no valid configuration possible), are automatically detected and reported to the user using the* Problems *view.*

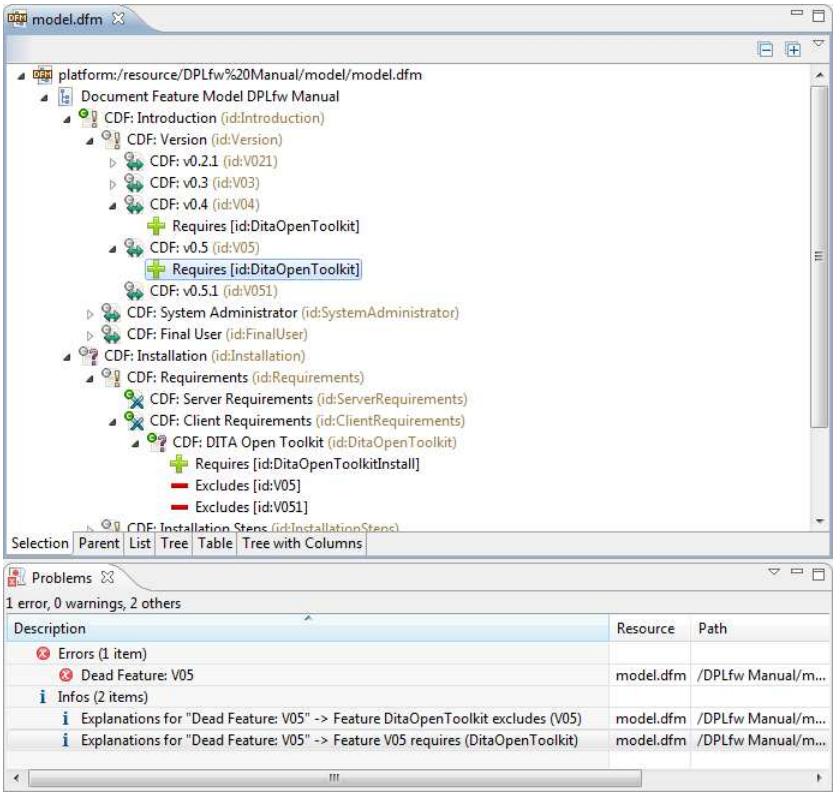


Figure 10: Feature Editor and Problems View

5.4. The Configuration Editor

Configurations are defined with the *Configuration Editor*, which guides users through the first task of the *Application Engineering* stage: the characterization of a specific document. The editor relies heavily on the *Validation Module*, which checks the model on every user decision, allowing a staged configuration. This way, when a feature is selected, all its mandatory child features are automatically selected, whereas optional and alternative features must be selected manually. If the selected feature has unselected parents, they are automatically selected, too. When a configuration changes, the features that cannot be selected according to the model constraints are automatically "disabled". If any of these actions (manual or automatic) collide with a model constraint, and no automatic solution can be provided, the

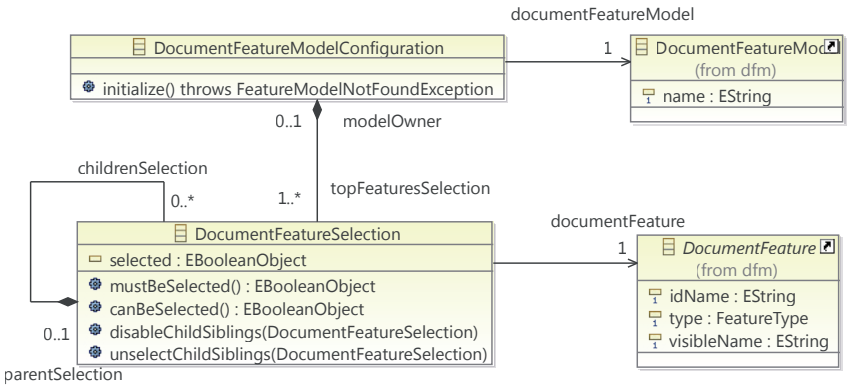


Figure 11: Document Configuration Metamodel

editor will report all the conflicting selections and ask the user for a corrective action. This scenario can occur, for example, when the model contains complex dependency or exclusion restrictions.

Every feature model configuration is stored as a separate artifact, which is linked to the document feature model. Figure 11 shows a scheme of the document configuration model. It consists of a *DocumentFeatureModelConfiguration* linked to a *DocumentFeatureModel*, and contains a hierarchy of *DocumentFeatureSelections* whose structure resembles the structure of the feature model. A *DocumentFeatureSelection* has a selected state which can have three possible values: `true` if the associated feature is selected, `false` if the associated feature is unselected, or `null` if the state of the feature has not been decided on.

In the case of feature model modifications, a configuration can be automatically updated with the changes made to the model. The features that remain after the update will keep their selection status, whereas the new ones will remain unselected. If the new features invalidate the previous configuration, the error reporting mechanisms will guide the user through the reconciliation tasks that must be performed. These tasks must be done manually since there is no previous information available.

Example 9 *The application engineering stage exploits the variability model to generate the final document. In our study, the document engineer uses the* Configuration Editor *to characterize the document to be generated for the DPLfw manual. In this case, to illustrate the generation process with a*

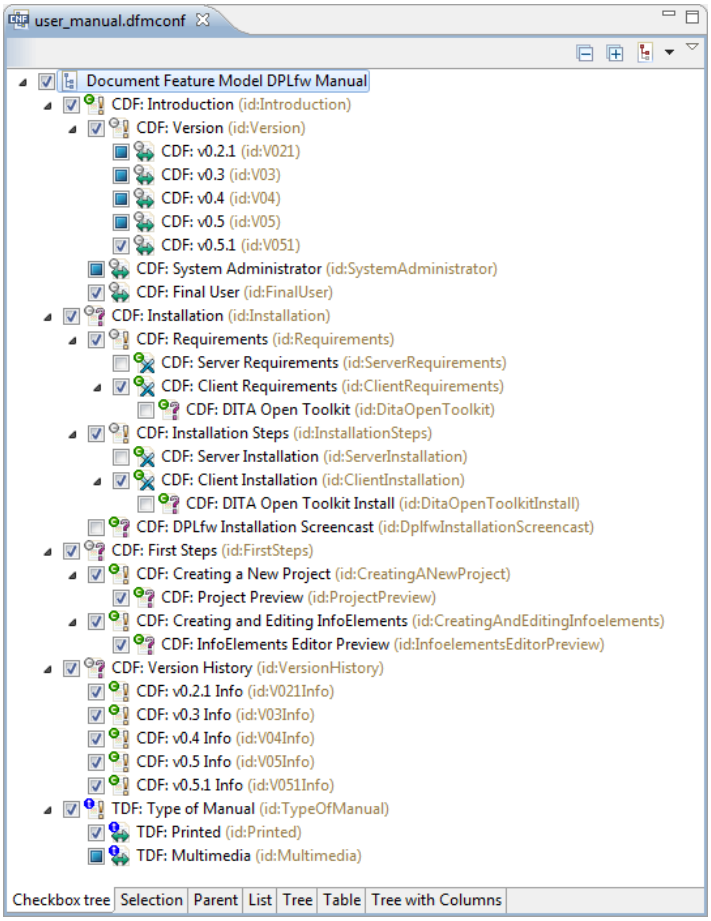


Figure 12: Configuration Editor

meaningful example, the document engineer decides to create a manual for advanced end users. Figure 12 shows the document configuration representing a printed manual of DPLfw *v0.5.1 for advanced users. This manual will contain the information for regular end users together with some additional contents (i.e.* Version History *and* Installation *of the* DPLfw *client). The editor ensures that all the model restrictions are met. For example, when the* Final User *feature is selected, the required feature* First Steps *must also be checked. The editor also guarantees that features related to the* DITA Open Toolkit *cannot be checked, since they are excluded for Version 0.5.1. Once the CDFs are selected, the TDF* Type of Manual *is selected as* Printed.

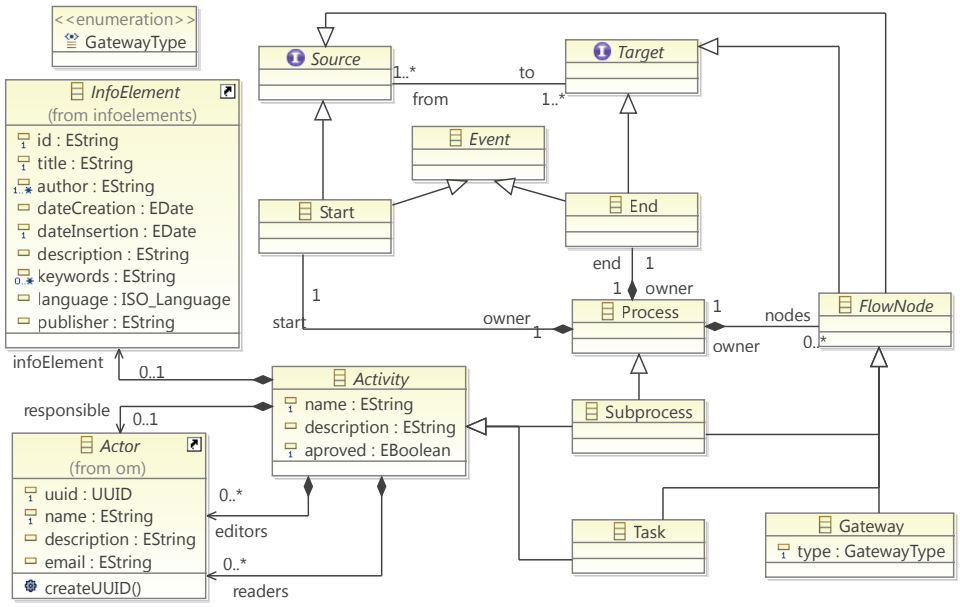


Figure 13: Workflow metamodel

5.5. The Workflow Editor

For a given document configuration, a document workflow model is automatically generated from the relationships between the CDFs. The workflow model is an instance of the metamodel shown in Figure 13, which is based on BPMN [21]. The document workflow metamodel describes a *process* that has a beginning (*start* event), an end (*end* event), and a set of *activities* which are executed between these two events, according to a control flow. An *activity* may be a *task*, a *subprocess*, or a *gateway*. A *task* is an atomic activity which cannot be broken down to a finer level of detail. A *subprocess* is an activity whose internal details are modeled using *activities*, *gateways*, and *control flows*. A *gateway* is used to control how the flows converge (in a join gateway) and diverge (through a split gateway) within a process. Finally, *FlowNode* is used to provide a single element as the source and the target that can appear in a process flow (*tasks*, *subprocesses*, and *gateways*).

The generation of the document workflow model is as follows. For each CDF of the document configuration, an activity is added to the workflow model. In order to preserve the content of the *InfoElement* in the repository,

a copy of it is created and assigned to the activity; the assignment of actors to the CDF is also propagated to the activity via the associations with the same names used in the Features Metamodel (i.e., *responsible*, *editors* and *readers*). CDF actors and permissions are copied and assigned to the corresponding activity. If a CDF has no subfeatures, a task is created, otherwise, a subprocess is created instead, i.e. the activity is broken down into several subactivities.

The different activities are ordered according to a control flow specification, which may be derived from the relationships defined in the document feature model. For instance, if an activity needs the value of some data generated by another activity, the former cannot be performed before the completion of the latter. Additionally, different patterns may be applied to organize the activities of the process (or subprocess) generated. These patterns depend on the TDF, i.e. the media of the final document. For example, for printed media, a sequence of activities is generated according to the order of the corresponding CDFs located at the same level in the document feature model. For multimedia, a set of parallel activities is generated. The automatically generated document workflow model may be modified using the workflow editor that has been added to DPLfw.

Example 10 *A* document workflow model *is automatically generated using the previously defined document configuration. The workflow editor added to DPLfw shows the creation workflow of the DPLfw manual for advanced users. This document workflow has four sequential activities which correspond to the four selected top-level CDFs. The workflow uses a sequential pattern because the final document will be a printed document (although other patterns may be applied). The four activities are subprocesses (i.e., complex activities), and may be expanded to show their internal activities. The* First Steps *subprocess has been expanded and is composed of two tasks: the first one is associated with the* Creating a New Project *CDF and the second with the* Creating and Editing InfoElements *CDF. Simple activities, such as the* Project Preview*, are called tasks and cannot be expanded. The actors associated with each activity can also be shown. In the figure only the* Responsible *and* Editors *of the* Project Preview *tasks have been expanded for reasons of clarity. Subprocesses may be also opened in another editor window, enhancing the scope of the edition and decreasing the complexity of the graphical representation. Finally, it is worth mentioning that the document engineer*

may modify the whole workflow using the tool palette provided by the workflow editor (top-right in Figure 14).

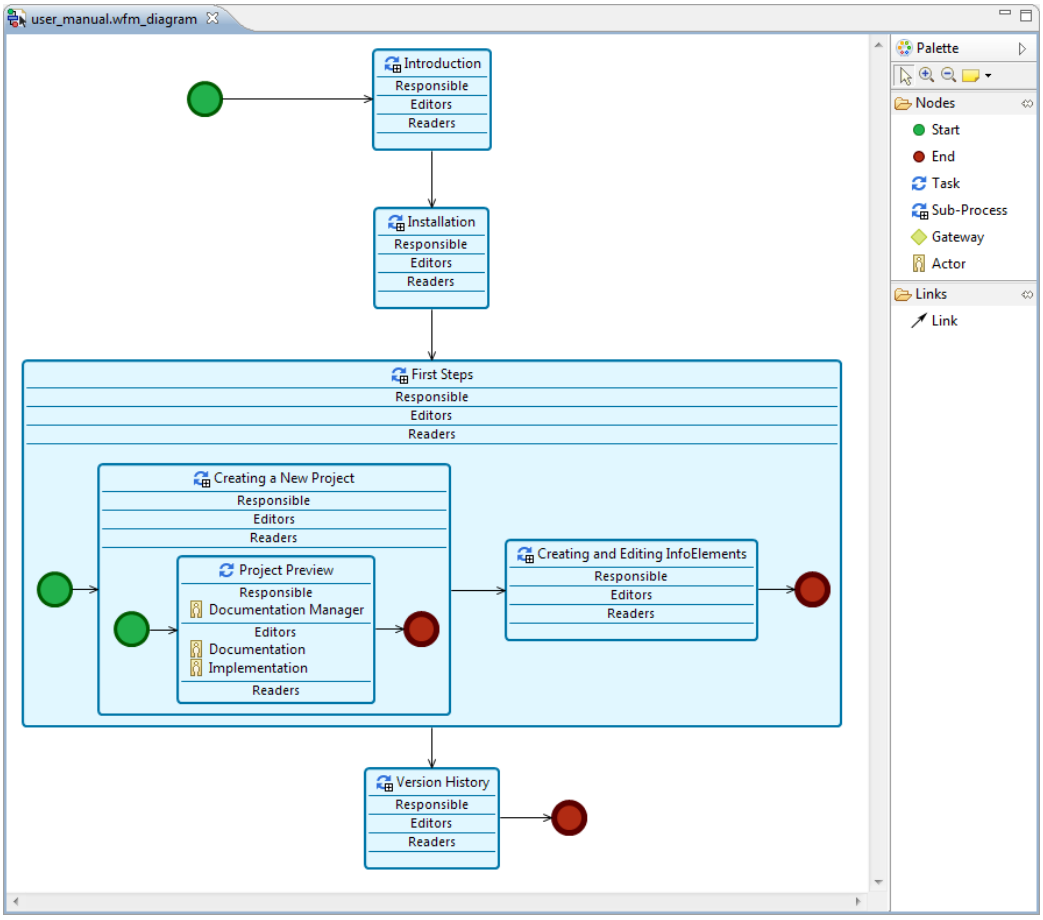


Figure 14: Document Workflow Editor

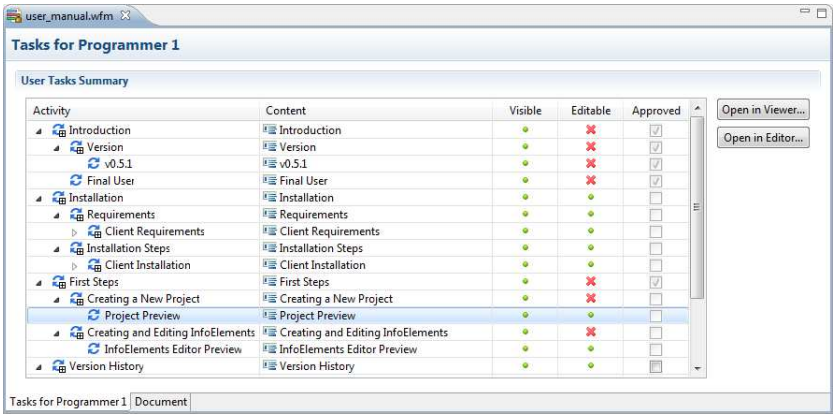
5.6. Custom Document Editors

The *Custom Document Editors* are the software components used to enact the previously defined document creation workflow which produces the final document content. These editors provide both a task-oriented and user-centered view of the document based on whatever editing tasks and permissions they have been given.

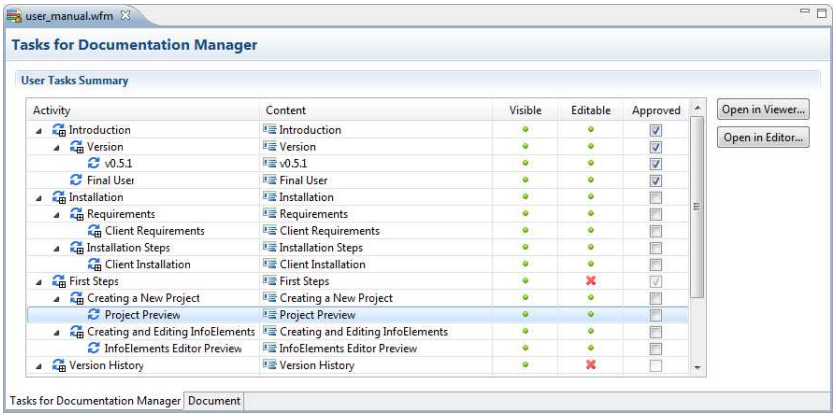
At this point, the document is stored as an encrypted resource that can only be edited by the custom editors, and users are required to introduce their credentials to view/modify/approve document contents. To complete his/her editing tasks, a user must log into the system using his/her login and password (which are validated against the *Credentials Manager*). Once the credentials are successfully validated, two different views are presented to the user: first, a list of all the tasks (and their corresponding *InfoElements*) that require the user's attention; and second, a (possibly partial) preview of the document using the predefined disseminators assigned to each type of *InfoElement*. These disseminators may allow the edition of the document content based on the user permissions.

Example 11 *Figures 15 and 16 show the two different views provided by the custom editors. Figure 15a shows the tasks (column* Activity*) and permissions (columns* Visible*,* Editable *and* Approved*) given to* Programmer 1*, the manager of the development unit. Figure 15b shows the tasks assigned to the* Documentation Manager*. A green circle in the* Visible *and* Editable *columns means that the associated* InfoElement *can be viewed/edited, otherwise, a red cross is used. Checkboxes on the* Approved *column are enabled according to whether or not the current actor is responsible for the task. Non-editable* InfoElements *such as* First Steps *are approved by default.*

Multiple actors may have permission to contribute to the same InfoElement *associated with a task. For instance, the* Project Preview *may be read and edited by* Programmer 1 *or the* Documentation Manager*; however the approval of its content is the responsibility of the* Documentation Manager *only (it can be seen that* Programmer 1 *is not allowed to change its state). Once a task has been approved its associated* InfoElement *cannot be edited further (unless the user has approval rights), as can be seen in the tasks associated with the* Introduction*. Once all the tasks have been approved, the workflow has been completely enacted.*



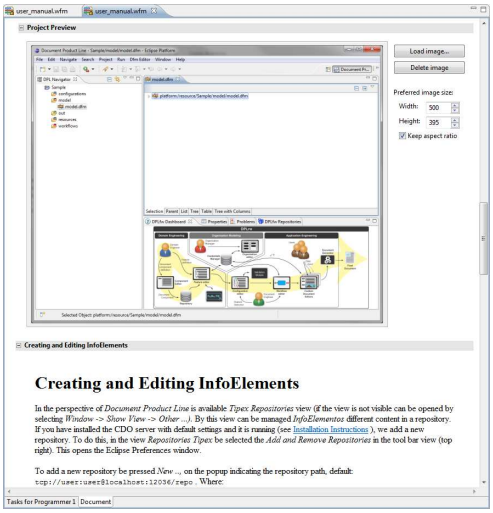
(a) Task list for *Programmer 1*



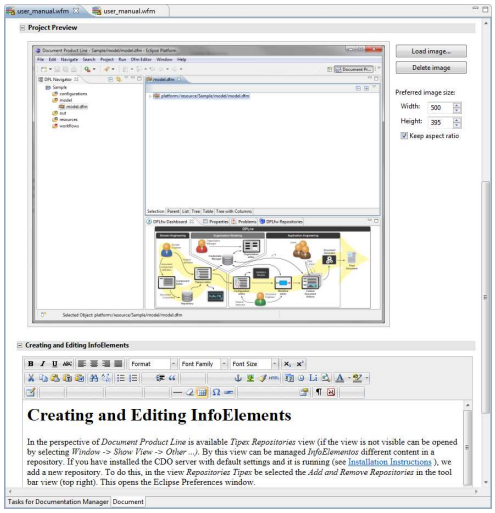
(b) Task list for *Documentation Manager*

Figure 15: Custom Document Editors (task list)

As mentioned previously, each actor has a customized view of the document which is closer to its final appearance. In this case, the Documentation Manager *has its own document editor containing all the visible contents. This custom view can not only be used to preview the document, but also to edit it: all the* InfoElements *that can be edited by an actor (and are not yet approved), are shown by the appropriate editing disseminator. Figure 16b shows an example of this case. Notice that both the* Project Preview *and* Creating and Editing InfoElements *can be edited by* Documentation Manager*. Figure 16a shows the view of the same part of the document for* Programmer 1*. In this case the* Project Preview *can also be changed, but the* Creating and Editing InfoElements *cannot.*



(a) Document Editor for *Programmer 1*



(b) Document Editor for *Documentation Manager*

Figure 16: Custom Document Editors (document preview)

5.7. Document Generation

Once all the users have provided their contributions to the document, and all the modifications have been approved by the person responsible, the *Document Generator* produces the final document.

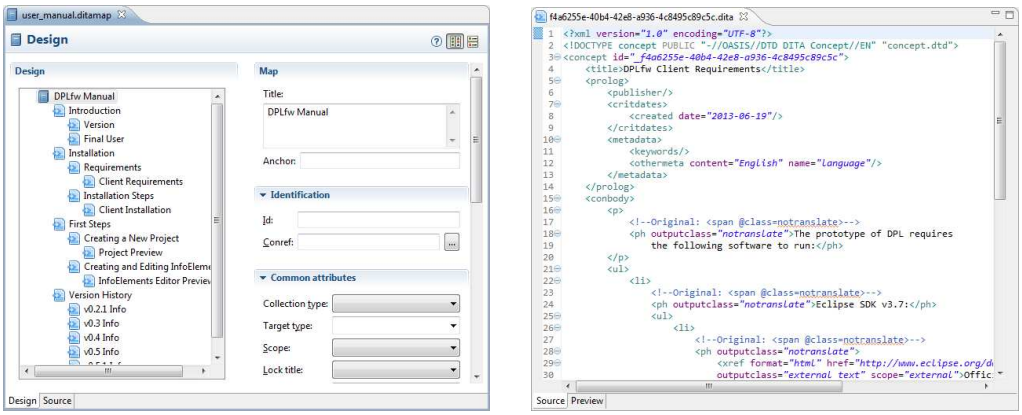
To implement the *Document Generator*, we have again used the DITA Open Tool Kit engine [40]. Since DITA topics have both data and metadata (a structure similar to that of *InfoElement*), we chose DITA as the implementation technology in early versions of DPLfw [7] and maintained the association for its good compatibility and tool support, which allows us to generate documents in a great variety of formats without having to worry about layout issues.

From a document workflow enacted by the custom editors it is easy to obtain a DITA specification which represents the final document: the structure of the map can be inferred from the structure of the editing activities, and the topics can be obtained from the corresponding *InfoElements*. The DITA map is used to generate the final document using the DITA Open Tool Kit engine. Before generating the final document, DPLfw still allows

some additional document customization using the DITA editors included in DPLfw. These editors are based on the ones provided by the DITA Open Platform Editor project [41].

Using the DITA-related tools, any variable-content document can be edited and generated in DPLfw, whether they are linked HTML documents, PDF files, Microsoft Word files, etc.

Example 12 *In our example, when all the actors have made their contributions, and they have been approved by the* Documentation Manager *or* Programmer 1*, the final document can be produced. This final step is done in two automated phases with minimal user interaction. First, a DITA specification (i.e. a DITA map and its topics) is automatically obtained from the enacted document workflow model as shown in Figure 17. Then, the integrated* DITA Open Tool Kit *engine uses the DITA specification to produce the final document in the selected format (in this case a PDF file as shown in Figure 18).*



(a) The DPLfw manual as a DITA map

(b) The "DPLfw Client Requirements" DITA topic

Figure 17: The DPLfw manual as a DITA specification

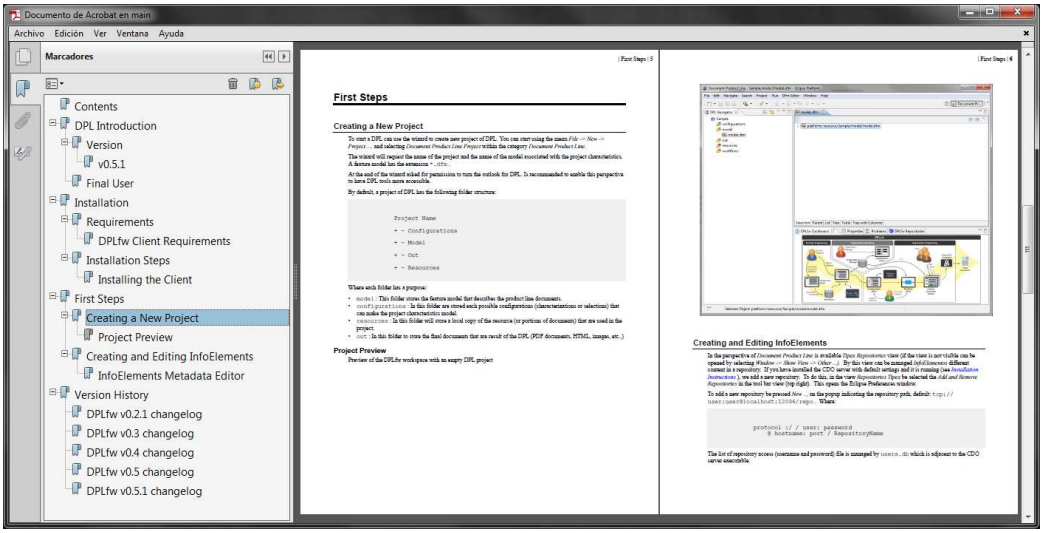


Figure 18: The final DPLfw manual for advanced end-users as a PDF file

6. Comparison of DPLfw with other Variable Content Frameworks

In this section we have grouped related works in two areas of research. One includes approaches on Variable Data Printing (VDP) and the other research on variable content document generation based on a product line approach. In this way we deliberately highlight the distance between VDP and product line-based solutions, the group to which DPLfw belongs.

6.1. Variable Data Printing

Among the most remarkable approaches to VDP, the Document Description Framework (DDF) proposed by Lumley et al. defines a document representation format based on three separate spaces: application data, logical data structure and presentation instructions. In [1] they propose an extensible architecture based on DDF to support the editing and authoring of sets of variable-data documents. DDF is flexible and extensible, its syntax is XML-based and the construction of new document parts from variable data is declared in embedded XSL templates. The document layout is declared through extensible functions in the presentation space. The language used to

define variability is powerful, but the use of embedded XSL makes the proposal lack flexibility and requires a high degree of expertise in XML-related technology.

Sellman [2] makes VDP work easier to design, and puts control of important aspects in the hands of designers, leading to document templates that are easy to alter and enhance as new requirements arise. These templates are versioned with content variables and layer variants and different themes can act independently or cooperatively to achieve rich but bounded layout variance. This approach uses PPML [42] to describe the layout of a document in terms of addressable objects, and each of the objects can be represented using different formats. Like DDF, the PPML document engineering architecture is based on XML and the final documents can be generated, merged, manipulated and processed using standard XML tools.

Piccoli et al. [6] propose an interactive authoring method for creating personalized free-form documents which is used for automatically distributing and manipulating images, text and decorative elements on a page. The proposal is essentially a semi-automatic method for document layout design, allowing the user to easily specify the desired layout. A simple authoring tool prototype was developed to test the proposed interaction model and produce a PDF file.

All these – and other similar – VDP proposals do not provide support to the multiple actors involved in editing tasks. Only PPCD [43] provides a differential access control to documents by multiple participants in cross-organizational workflows. The document circulates between workflow participants, who have to contribute to various parts of the document at different access levels (edit, read, etc.). A prototype authoring tool was developed that allows automatic and manual selection of workflow participants directly from an LDAP directory together with their access permissions.

Some commercial solutions also exist. PageFlex [4] is a business solution for variable publishing that includes tools for the design of document templates and a web version to customize and order documents online. Its main characteristics are: a content-driven approach which provides information in a web form, merging of data files with a design template and web browser-based user interaction. PlanetPress Suite [44] allows for easy creation of variable content documents with the added benefits of offering advanced automated workflow and output management features. The main goals are (i) to produce variable content business documents aligned with the business processes of the organization and (ii) optimize their distribution within the

organization, supporting a wide variety of data and document input (ascii, database, PDF, XML, etc.). Other similar solutions are DialogueLive [45] or FusionPro™[46].

6.2. Variable content document generation based on a product line approach

The proposal of Rabiser et al. [47] is an interesting case of early variability management in documents. They face the problem of aligning document content with variants in a product line. Using DOPLER, a decision-oriented Software Product Line environment, they implementat a flexible document generation process that aims to reduce the time-to-market of documents such as offers, catalogs, etc. and also to avoid inconsistencies in creating technical documentation associated with products with variable features. A methodology to support the development of variable content document is also included, although in general the main focus of the work is on aligning document content with variation points in the product line, and not on the production of documents as the final artifacts of the product line. The variability mechanism in DOPLER is based on DocBook [48], an XML documentation language designed specifically to produce technical documentation.

Karol et al. in [49] present an approach to specifying document variants in a family of office documents (ODF) using features models. FODA is used to specify variability in documents. Features are associated to specific documents or part of documents using a mapping editor. Variability is modeled by a *negative* approach [50], i.e., given a mapping and variant specification, an interpreter creates a copy of the documents involved and removes all unnecessary parts according to the features selected in the variant specification. The Document Feature Mapper prototype works on XML and ODF documents.

Another proposal for customizing business documents based on explicit variability models is [51]. The proposal is focused on the Core Component approach [52], a conceptual approach for defining business document types based on generic, reusable building blocks. A mapping between Core Components and cardinality-based feature models is performed, and feature models may be generated based on Core Components. The proposal explores different types of variability (additive and negative) and helps the user to describe business documents. Document generation (known as *business document derivation*) is based on a feature model configuration and model-driven generation to integrate the Core Components selected. A prototype is being developed to validate the proposal but document generation is not included. Other works focus on how to manage the evolution of business document

models, including the evolution of metamodels, feature models and feature model configurations, as well as co-evolution of business document instances [53] and the validation of the business document models specified [54]

6.3. Comparison and results

Table 2 shows a comparison of the present proposal with previous approaches. The commercial products are grouped in the same column, since they give similar performance and support for the different requirements studied in this paper. The comparison is based on the challenges outlined in Section 2, which have been broken down into ten topics: *Explicit Content Variability Modeling*, *Technology Variability Modeling*, *Variability Model Validation*, *Multiple Actors in Variability Model*, *Document Configuration*, *Document Workflow Generation*, *Generation of Customized Content Editor by Actor*, *Final Document Generation*, *Methodological Approach* and *Framework*. In Table 2, the labels could be: (i) ✗ (unsupported: according to our understanding of the proposal, the requirement is not fulfilled by the proposal); (ii) ✔ (partially supported: only some of the aspects of the requirement are fulfilled by the proposal); (iii) ✔ (supported: the requirement is fulfilled by the proposal and a solution is provided); and, (iv) **?** (not specified: the topic is not explicitly reported in the papers reviewed).

Explicit Content Variability Modeling All the proposals presented in Sections 6.1 and 6.2 support the generation of variable content documents, with the exception of Pichler et al. [51], which will be discussed below. Their main difference lies in whether or not content variability is explicitly represented. In the VDP proposals, XML-based document fragments are defined and the links between them and the transformation rules represent the variability model implicitly. However, Sellman [2] refines the concept of XML-based document fragments and introduces new XML elements to deal with variability (using themes and layer variants). On the other hand, the proposals based on product lines model variability explicitly from a domain-oriented perspective using feature models. Both Karol et al. [49] and DPLFW use FODA feature models, whereas Pichler et. al [51] use richer representations such as cardinality-based feature models. Rabiser et al. [47] do not use feature models, but decision models, in their DOPLER framework. The decisions are abstract representations of variation points in the assets model (the document fragments model).

	VDP APPROACHES					PRODUCT LINE APPROACHES			
	Lumley et al. [1]	Sellman [2]	Piccoli et al. [6]	Balisky et al. [43]	Commercial Products	Rabiser et al. [47]	Karol et al. [49]	Pichler et al. [51]	DPLFW
Explicit Content Variability Modeling	✔ (XML fragments)	✔ (themes & layer variants, XML based)	✔ (XML fragments)	?	✔ (document fragments)	✔ (assets & decisions model)	✔ (FODA)	✔ (cardinality-based)	✔ (FODA)
Technology Variability Modeling	✗	✗	✗	✗	✗	✗	✗	✗	✔
Variability Model Validation	✗	✗	✗	?	✗	?	✗	✔	✔
Multiple Actors in the Variability Model	✗	✗	✗	?	✗	✔ (for decision model)	✗	✗	✔ (for feature model)
Document Configuration	✔ (XML tools)	✔ (XML tools)	✔ (content selection)	?	✔ (content selection)	✔ (customized by actor)	✔	✗	✔
Document Workflow Generation	✗	✗	✗	✔	✔	✗	✗	✗	✔
Generation of Customized Content Editors	✗	✗	✗	✔	✔	✗	✗	✗	✔
Final Document Generation	✔ (w/ layout)	✔ (w/ layout)	✔ (w/ layout)	✔ (w/o layout)	✔ (w/ layout)	✔ (w/o layout)	✔ (w/o layout)	✗	✔ (w/o layout)
Methodological Approach	✔ (tool-driven)	✔ (tool-driven)	✔ (tool-driven)	✗	✔ (tool-driven)	✔	✔ (tool-driven)	✗	✔ (DPL)
Framework	✔ (generic XML tools)	✔ (Design Maker, based on XML tools)	✔ (prototype)	✔ (prototype)	✔ (commercial product)	✔ (DOPLER)	✔ (Document Feature Mapper)	✗	✔ (DPLFW)

Table 2: Comparison of proposals for variable content document generation

Technology variability modeling This topic decides if the proposal allows technology variability to be modeled explicitly. Technology variability in DPL is defined as that used to represent document contents. According to our understanding, none of the reviewed proposals except DPLFW represent technology variability from a domain-oriented perspective. In the other proposals the link between content document fragments and how they are rendered in the final document is managed implicitly using XML stylesheets. Although the final document may be generated in different formats (PDF, HTML, ODF, MS Word, etc.) this feature is not explicitly described. It is noteworthy that Piccoli et al. [6] do not support this and that only PDF files can be generated in their approach.

Variability Model Validation For proposals supporting explicit variability modeling, this topic detects whether the proposal provides automatic model validation capabilities to detect different types of error and provide error messages. Pichler et al. [54] provide some validation mechanisms to ensure that and changes made to the business document models are valid. DPLFW provides validation and verification mechanisms to ensure the validity of both feature models and feature model configurations. For feature models verification DPLFW relies on FAMA [33] and on the built-in validation module for feature model configuration validation. The other proposals reviewed do not include a module or mechanism to perform automatic model validation.

Multiple Actors in the Variability Model This topic identifies the enrichment of the variability model with the possibility of including multiple actors. Actors may have different rights and responsibilities in the specification of the document fragments and variability model, as well as in the generation of the final document. The VDP proposals do not support this requirement because they do not provide an explicit content variability model. DOPLER [47] allows actors and roles to be specified in the decision models, so that responsibility for decisions in the product configuration stage can be assigned and enforced. On the other hand, DPLFW allows the actors who play a role in creating the document to be specified, while the roles in DPLFW are related to document contents and not to document configuration (selection of features).

Document Configuration This topic relates to the selection of variability points to generate a member of the document of the family (a variant). All the proposals support this requirement, except that by Pichler et al. [51] (whose main goal is the specification and validation of variants of the Core Component standard using feature models, not the variable content document generation itself). VDP proposals support document configuration with XML-related tools, which select the document fragments and combine them into a final document. Piccoli et al. [6] and the commercial products reviewed additionally provide a user-friendly interface for selecting variable content (using forms or web forms). In the product line-based proposals there is an important difference between DOPLER [47] and DPLFW. DOPLER provides a configuration wizard for document characterization which is performed according to the decision model. Since it contains information about actors, the wizard only shows the decisions in which the user is involved. In DPLFW, the configuration editor provides a single view so that the document can be characterized by a single actor (the Document Engineer), who selects the variants for each variability point.

Document Workflow Generation This topic covers whether the proposal is able to represent the document generation process explicitly and if this is automatically generated. The task flow is specified in the document workflow as well as the different actors (and their access rights) that contribute to the final document. In the VDP proposals this requirement is only supported by Balinsky et al. [43] and the commercial products. The former is a specific proposal to create and manage multipart composite documents, where participants (including external consultants, partners and customers) interact in a secure and distributed environment. In the commercial products the document workflow is partially supported by the product itself, which allows different actors to participate although the document workflow is not represented explicitly. Regarding the product line-based proposals, only DPLFW supports the generation of document workflows as an explicit artifact which represents the whole process, where actors and access rights are explicitly represented and tuning permissions and rearranging tasks are allowed.

Customized Content Editors by Actor This topic is related to the pre-

ceding one and determines whether the system generates custom editors that represent a user-centered view of the document workflow. These custom editors are used by the different actors to contribute to the document contents according to their responsibilities and access rights. Both Balinsky et al. [43] and DPLFW support the generation of these custom editors. Commercial products only support this partially, since the workflow is implicitly represented and managed by the tool (i.e., it is product-driven).

Final Document Generation All the approaches in Table 2 support the generation of a final document, in accordance with the variability selected in the document configuration. The main difference between the proposals is that not all allow the definition of custom document layouts. As mentioned in Section 5.3, a document is the union of content and presentation. The presentation includes the layout, which defines exactly where each piece of content is to be placed and also how the piece will appear in the document. The VDP proposals support the layout applied in document generation, except in Balinsky et al. [43], in which the layout is orthogonal to the PPCD proposal. However, the layout is not considered in the proposals based on product lines, although their support is also orthogonal. DPLFW does not consider layout in document generation.

Methodological Approach This topic determines whether the proposal includes methodological guidance to the generation of documents with variable content. Most of the VDP proposals do not provide explicit guidance, but rather the tool functionality guides the process. Regarding the product-line-based approaches, Balinsky et al. [43] and Pichler et al. [51] do not provide methodological guidelines for document generation, unlike DOPLER and DPLFW. DPLFW is based on the DPL methodology.

Framework Finally, the last requirement determines whether the proposal is supported by a specific tool or framework.

To sum up, as discussed above and as Table 2 shows, most of the aforementioned VDP proposals and tools are presentation-oriented authoring and/or editing tools that focus on the final document, identifying the editable parts and how this information passes through intermediate processing to end up

on the final document. In most of them, XML technology supports editing and transformation of the documents via stylesheets. However, none of them has paid attention to providing methodological guidance to, for instance, identifying variable parts or to managing variability at the requirements level to improve traceability and reduce potential inconsistencies during document generation. Additionally, support for multiple actors is rarely provided, except in Balinsky et al., who propose a general solution for composite documents. Commercial products are also focused on the final document and as such, their solutions are tool-driven (i.e. content selection and document workflow management are driven by the navigation of a web-based application or similar).

Proposals based on product line principles, however, do provide methodological guidance to model the variability of a family of documents. The domain and application engineering stages guide document generation with or without participants. The emphasis is on the definition of a process to supply reuse and automation in document generation and not in the layout of the final document. The closest approach to DPLFW is the proposal by Karol et al. However, in the latter there is no distinction between content and technology features and (unlike DPLFW) no support for multiple participants is provided. Finally, this proposal is targeted at office applications, while DPLFW is a general purpose tool. On the other hand, DOPLER and DPLFW do have topics in common, but DOPLER does not use feature models to manage variability. Furthermore, inclusion of multiple participants is intended for document configuration, whereas in DPLFW it is intended to generate customized content editors. Another difference not shown in Table 2 is that DOPLER document generation is based on DocBook [48], while DPLFW uses DITA [36]. Both DocBook and DITA are XML documentation languages designed specifically to produce technical documentation, but there are some differences. DITA is topic-oriented (separates content from the use context), while DocBook is fragment-oriented (content is related to chapters, sections, paragraphs, etc.). In addition, DITA is extensible, allowing information types to be defined, while DocBook has a fixed element and attribute set. Finally, the Pichler et al. proposal has similarities to DPLFW; both use feature models to identify variability points and validate the models for checking and detecting errors, but their main goal is different. As mentioned above, Pichler et al. do not support the generation of final documents, an important topic, as document workflow is supported by the Balinsky et al. proposal and DPLFW. However, the former is a single-document-oriented

solution and DPLfw supports families of documents following a product line approach. Finally, we could not find any proposal that allowed explicit content and technology variability modeling.

7. Conclusions and future work

In markets with millions of potential users, generating personalized document versions is unaffordable unless tools supporting automation and content reuse are available. In this paper, we introduce DPLfw, a framework and tool supporting the DPL methodology for multi-user, variable content and reuse-based document generation. A DPL process starts with the development of a document feature model that defines the characteristics of a family of documents and the contributors involved (actors). A specific document (an instance of the family) is then created following a process in which the document components are taken from a repository, maximizing reuse. While previous approaches for variable content document generation are presentation and technology-oriented (i.e. they focus on the final document), DPL captures variability in the early stages of the domain engineering phase and focuses on providing methodological guidance.

DPLfw is a fully functional tool which covers the whole DPL document lifecycle, from analysis of the domain (document family) to the generation of the final document. Besides its basic function, DPLfw provides advanced support for document feature model checking and validation, helping users to avoid mistakes in defining both feature models and configurations. Another important feature which distinguishes DPLfw from its competitors is the support it provides for the generation of custom document editors, allowing multiple actors to contribute to complex documents, focusing only on the tasks they are involved in. Additionally, content description at the domain level can ease document design tasks since no knowledge of low level markup languages is required.

DPLfw has been used in different variable content case studies, including the generation of software manuals, drawing up emergency plans, documenting software processes and generating recipe families. Details on these case studies have been published elsewhere [7, 8, 10, 55, 56, 57]. However, a thorough analysis of the tool must still be performed and is the main focus of our further work as the final part of the TIPEx research project [58]. Other items on our research agenda include the enrichment of the document workflow metamodel to include new tasks. We are also studying other control flow

patterns to generate more sophisticated document workflow models. Finally, we are planning to develop a DPLFW plugin providing a fully collaborative edition environment, where multiple actors can contribute concurrently and interactively to document content.

8. Acknowledgments

This work has been partially funded by the Spanish Ministerio de Educación y Ciencia under grant TIPEx (TIN2010-19859-C03-03).

References

[1] J. Lumley, R. Gimson, O. Rees, A framework for structure, layout & function in documents, in: Proceedings of the 2005 ACM symposium on Document engineering, DocEng '05, ACM, New York, NY, USA, 2005, pp. 32–41.

[2] R. Sellman, VDP templates with theme-driven layer variants, in: Proceedings of the 2007 ACM symposium on Document engineering, DocEng '07, ACM, New York, NY, USA, 2007, pp. 53–55.

[3] Mail Merge in Wikipedia, 2011. URL: `http://en.wikipedia.org/wiki/Mail_merge`.

[4] Bitstream Inc., Pageflex, 2012. URL: `http://www.pageflex.com/`.

[5] N. Hurst, W. Li, K. Marriott, Review of automatic document formatting, in: Proceedings of the 9th ACM symposium on Document engineering, DocEng '09, ACM, NY, USA, 2009, pp. 99–108.

[6] R. F. B. Piccoli, R. Chamun, N. C. Cogo, J. a. B. S. de Oliveira, I. H. Manssour, A novel physics-based interaction model for free document layout, in: Proceedings of the 11th ACM symposium on Document engineering, DocEng '11, ACM, New York, NY, USA, 2011, pp. 153–162.

[7] M. C. Penadés, J. H. Canós, M. R. Borges, M. Llavador, Document product lines: variability-driven document generation, in: Proceedings of the 10th ACM symposium on Document engineering, DocEng '10, ACM, New York, NY, USA, 2010, pp. 203–206. URL: `http://doi.acm.org/10.1145/1860559.1860603`.

[8] A. Gómez, M. C. Penadés, J. H. Canós, M. R. S. Borges, M. Llavador, Dplfw: a framework for variable content document generation, in: Proceedings of the 16th International Software Product Line Conference - Volume 1, SPLC '12, ACM, New York, NY, USA, 2012, pp. 96–105.

[9] S. Kent, Model driven engineering, in: M. J. Butler, L. Petre, K. Sere (Eds.), Integrated Formal Methods, Third International Conference, IFM 2002, Turku, Finland, May 15-18, 2002, Proceedings, volume 2335 of *Lecture Notes in Computer Science*, Springer, 2002, pp. 286–298.

[10] M. C. Penadés, A. Gómez, J. H. Canós, Deriving document workflows from feature models, in: Proceedings of the 2012 ACM symposium on Document engineering, DocEng '12, ACM, New York, NY, USA, 2012, pp. 237–240.

[11] C. F. Goldfarb, The SGML Handbook, Oxford University Press, Oxford, UK, 1990.

[12] D. Parnas, On the design and development of program families, Software Engineering, IEEE Transactions on SE-2 (1976) 1–9.

[13] P. Clements, L. Northrop, Software Product Lines: practices and patterns, volume 0201703327, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[14] Workflow Management Coalition, Workflow Management Coalition website, 2012. URL: `http://www.wfmc.org/`.

[15] M. Havey, Essential Business Process Modeling, O'Reilly Media, Inc., 2005.

[16] L. Chen, M. A. Babar, N. Ali, Variability management in software product lines: A systematic review, in: Proceedings of the 13th International Software Product Lines Conference (SPLC'09), San Francisco, CA, USA.

[17] K. C. Kang, Keynote Address: FODA: Twenty Years of Perspective on Feature Models, in: Software Product Line Conference (SPLC) 2009, USA.

[18] K. Kang, S. Cohen, J. Hess, W. Nowak, S. Peterson, Feature-Oriented Domain Analysis (FODA) Feasibility Study, 1990.

[19] D. Batory, D. Benavides, A. Ruiz-Cortés, Automated analysis of feature models: Challenges ahead, Communications of the ACM December (2006).

[20] D. Benavides, S. Segura, A. Ruiz-Cortés, Automated analysis of feature models 20 years later: A literature review, Information Systems 35 (2010) 615 – 636.

[21] OMG, Business Process Model and Notation (BPMN) Version 2.0 (formal/2011-01-03), 2011. URL: http://www.omg.org/spec/BPMN/2.0/.

[22] OMG, MDA Guide Version 1.0.1. (omg/2003-06-01), 2003. URL: http://www.omg.org/cgi-bin/doc?omg/03-06-01.

[23] Eclipse Foundation, Eclipse platform technical overview, 2003. URL: http://www.eclipse.org/whitepapers/eclipse-overview.pdf.

[24] J. McAffer, P. VanderLei, S. Archer, OSGi and Equinox: Creating Highly Modular Java Systems, Eclipse series, Addison-Wesley, 2009.

[25] Eclipse Foundation, Eclipse Modeling Framework Project (EMF), 2012. URL: http://www.eclipse.org/modeling/emf/.

[26] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks, EMF: Eclipse Modeling Framework, Addison-Wesley, 2nd edition edition, 2009.

[27] Eclipse Foundation, The CDO Model Repository (CDO), 2012. URL: http://www.eclipse.org/cdo/.

[28] OSGi Alliance, OSGi Service Platform Core Specification, Technical Report, OSGi Alliance, 2008. URL: http://www.osgi.org/Specifications/.

[29] PostgreSQL Global Development Group, PostgreSQL, 2012. URL: http://www.postgresql.org/.

[30] M. C. Penadés, J. H. Canós, S. Camarasa, M. R. Borges, A. S. Vivacqua, Generación de documentos basada en líneas de producto (document generation based on product lines), Actas XVI Jornadas sobre Ingeniería del Software y Bases de Datos. JISBD'2011. A Coruña, Spain. (2011). In spanish.

[31] D. Beuche, Modeling and building software product lines with pure: : variants, in: SPLC (2), Kindai Kagaku Sha Co. Ltd., Japan, 2007, pp. 143–144.

[32] J. McAffer, J.-M. Lemieux, Eclipse Rich Client Platform: Designing, Coding, and Packaging Java(TM) Applications, Addison-Wesley, 2005.

[33] ISA Research Group, FaMa Tool Suite, 2012. URL: http://www.isa.us.es/fama/.

[34] T. Berners-Lee, R. Fielding, L. Masinter, RFC 3986, Uniform Resource Identifier (URI): Generic syntax, 2005. URL: http://tools.ietf.org/html/rfc3986.

[35] A. Powell, M. Nilsson, A. Naeve, P. Johnston, T. Baker, Dublin core metadata initiative abstract model, 2007. URL: http://dublincore.org/documents/abstract-model.

[36] OASIS, Darwin Information Typing Architecture (DITA) Version 1.2, 2010. URL: http://docs.oasis-open.org/dita/v1.2/spec/DITA1.2-spec.html.

[37] E. Tsang, Foundations of Constraint Satisfaction, Academic Press, 1995.

[38] D. Benavides, On The Automated Analysis Of Software Product Lines Using Feature Models A Framework For Developing Automated Tool Support, Ph.D. thesis, University of Seville, 2007. URL: http://www.lsi.us.es/dbc/dbc_archivos/pubs/benavides07-phd.pdf.

[39] R. Kahn, R. Wilensky, A framework for distributed digital object services, Int. J. Digit. Libr. 6 (2006) 115–123.

[40] Anderson, Robert D and Shen, Jian Le and Elovirta, Jarno and Sirois, Eric and Weiser, Reuven, DITA Open Toolkit, 2012. URL: http://dita-ot.sourceforge.net/.

[41] Vedovini, Claude, DITA Open Platform Editor, 2010. URL: http://www.dita-op.org/.

[42] PODi: the Digital Printing initiative, Personalized Print Markup Language (PPML), 2012. URL: http://www.standards.podi.org/ppml/ppml-overview.html.

[43] H. Balinsky, L. Chen, S. J. Simske, Publicly posted composite documents in variably ordered workflows, IEEE TrustCom/IEEE ICESS/FCST, International Joint Conference of 0 (2011) 631–638.

[44] Objectif Lune Inc., PlanetPress Suite, 2012. URL: http://www.objectiflune.com/OL/Products/PlanetPressSuite/.

[45] Hewlett-Packard Development Company, L.P., HP Dialogue Live Software, 2012. URL: http://welcome.hp.com/country/us/en/prodserv/software/eda/products/dialogue-live.html.

[46] PTI Marketing Technologies™, The FusionPro™Family, 2012. URL: http://www.pti.com/datasheets/.

[47] R. Rabiser, W. Heider, C. Elsner, M. Lehofer, P. Grünbacher, C. Schwanninger, A flexible approach for generating product-specific documents in product lines, in: J. Bosch, J. Lee (Eds.), SPLC, volume 6287 of *Lecture Notes in Computer Science*, Springer, 2010, pp. 47–61.

[48] N. Walsh, R. Hamilton, DocBook 5: The Definitive Guide, Definitive Guide Series, O'Reilly Media, 2010.

[49] S. Karol, M. Heinzerling, F. Heidenreich, U. Assmann, Using feature models for creating families of documents, in: Proceedings of the 10th ACM symposium on Document engineering, DocEng '10, ACM, New York, NY, USA, 2010, pp. 259–262.

[50] I. Groher, M. Voelter, Expressing Feature-Based Variability in Structural Models, in: Workshop on Managing Variability for Software Product Lines.

[51] C. Pichler, C. Huemer, Feature modeling for business document models, in: Proceedings of the 15th International Software Product Line Conference, Volume 2, SPLC '11, ACM, New York, NY, USA, 2011, pp. 3:1–3:8.

[52] United Nations/Centre for Trade Facilitation and Electronic Business (UN/CEFACT), Core Components Technical Specification Version 3.0, 2009. URL: http://www.unece.org/cefact/codesfortrade/ccts_index.html.

[53] C. Pichler, M. Seidl, C. Huemer, Managing variability and evolution of business document models, in: Proceedings of the 2nd International Workshop on Model-driven Product Line Engineering (MDPLE 2010), MDPLE '10, CEUR Workshop Proceedings, 2010, pp. 61–72.

[54] C. Pichler, R. Engel, C. Huemer, Validation of business document types based on feature models, in: Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems, VaMoS '12, ACM, New York, NY, USA, 2012, pp. 27–36.

[55] M. C. Penadés, J. H. Canós, M. R. Borges, A. S. Vivacqua, A Product Line Approach to the Development of Advanced Emergency Plans, in: Proceedings of the 8th International ISCRAM Conference.

[56] A. Gómez Llana, M. Penadés Gramaje, J. H. Canós Cerdá, Generación de Documentos con Contenido Variable en DPLfw, in: Actas de las XVII Jornadas de Ingeniería del Software y Bases de Datos, pp. 629 - 642. ISBN 978-84-15487-28-9.

[57] J. H. Canós, M. C. Penadés, M. R. Borges, A. Gómez, A product line approach to customized recipe generation, in: Proceedings of the 5th international workshop on Multimedia for cooking & eating activities, CEA '13, ACM, New York, NY, USA, 2013, pp. 69–74.

[58] I. Aedo, V. Bañuls, J. H. Canós, P. Díaz, R. Hiltz, Information Technologies for Emergency Planning and Training, in: Proceedings of the 8th International ISCRAM Conference.